The Procedural Approach

- System is organized around procedures.
- Procedures send data to each other.
- Procedures and data are clearly separated.
- Focus on data structures, algorithms and sequencing of steps.
- Procedures are often hard to reuse.
- Lack of expressive and powerful visual modeling techniques.
- Transformation of concepts between analysis & implementation.
- Design models are a long step from implementation.

The Object-Oriented Approach..

- System is organized around objects.
- Objects send messages (procedure calls) to each other.
- Related data and behavior are tied together in objects.
- Modeling of the domain as objects so that the
- Implementation naturally reflects the problem at hand.
- Visual models are expressive and relatively easy to comprehend.
- Focus on responsibilities & interfaces before implementation.
- Powerful concepts: interfaces, abstraction, encapsulation, inheritance, aggregation & polymorphism.
- Visual models of the problem evolve into models of the solution.
- Design models are only a small step from implementation.
- Software is very complex - strive to reduce complexity!

Modeling

- Modeling enables better communication and visualization *for all stakeholders*. Successful OO designs almost always begin with a visual "object model" of the problem domain, involving both the domain experts and software designers.
- Would you let a contractor build your new house without blueprints?
- The essence of modeling is to show all *pertinent* detail.

Terminology

- ***Object*:** A thing that exists in the domain of the problem
  - E.g. An office building might have a number of 'Elevators', 'Offices', 'Security Doors', etc.
- Software is *object-oriented* if the design and implementation of that software is based on the interaction between the objects of the domain

- ***Class***: A template used for the creation of objects
  - A class describes various attributes an object might have
- E.g. A 'Person' class might have various attributes, including age, weight, height, eye colour, address, etc.

Object Model

Object-oriented technology is built upon a sound engineering foundation, whose elements we collectively called the *object model.*
The object model encompasses the principles of

- Abstraction
- Encapsulation
- Modularity
- Hierarchy
- Typing
- Concurrency and
- Persistence

Foundations of the Object Model

- The structured design method uses algorithms to build a complex system as fundamental building blocks
- Object-oriented design methods have evolved to help developers exploit the expressive power of object-based and object-oriented programming languages using the **class** and **object** as basic building blocks

OOA,OOD and OOP

- During the object-oriented analysis there is an emphasis on finding and describing the objects or concepts in the problem domain
  - ***Focus on* what *the system must do*. *Do the right thing***
- During object-oriented design, there is an emphasis on defining software objects and how they collaborate to fulfill requirements.
  - *Focus on how the system will do it.* Do the thing right.
- Analysis - investigation of the problem (what)
- Design - logical solution to fulfill the requirements (how)

How are OOP, OOA and OOD related?

- The products of object-oriented analysis serves as the models from which we may start an object-oriented design
- The products of the object-oriented design cant hen be used as a blueprints for completely implementing a system using object-oriented programming methods

The building blocks of object orientation

- Objects
- Methods
- Messages

Elements of Object Model

There are four ***major*** and three minor elements of object model. The **four major** elements of object model are:

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

- By *major*, we mean that a model without any one of these elements is not an object model or object-oriented

The three ***minor*** elements of the object model are:

- Typing
- Concurrency
- Persistence

- By *minor*, we mean that each of these elements is a useful, but not essential part of the object model.

Abstraction

An abstraction denotes the essential characteristics of an object that distinguish it from all kinds of objects and thus provide crisply defined conceptual boundaries, relative to the prospective of the viewer.

- Abstraction focuses upon the essential characteristics of some object, relative to the prospective of the viewer.
- Abstraction focuses upon the observable behavior of an object.

Types of abstraction

- **Entity abstraction**: An object that represents a useful model of a problem-domain or solution domain.
- **Action abstraction**: An object that provides a generalized set of operations, all of which perform the same kind of function.
- **Virtual machine abstraction:** An object that groups together operations that are all used by some superior level of control or operations that all use some junior-level set of operations.
- **Coincidental abstraction:** An object that packages a set of operations that have no relation to each other

Abstraction Modes

Software engineers focus on three primary modes of abstraction.

- Procedural abstraction
  - Focusing on the logical properties of an action
  - Ignoring the details of the action's implementation

- Data abstraction
  - Focusing on the logical properties of data
  - Ignoring the details of the data's representation

- Class Abstraction
  - Procedural abstraction + data abstraction

Encapsulation

"Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation"

- Encapsulation hides the details of the implementation of an object.
- Abstraction and encapsulation are complementary concept abstraction focuses upon the observable behavior of an object where as encapsulation focuses upon the implementation of that gives rise to this behavior

Modularity

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

- Modularity reduces the complexity into some degree.
- The modularity packages abstractions into discrete units.
- *Modularity helps us by giving a way to cluster logically related abstractions*
- *Modules serves as the physical containers in which we declare classes and objects of our logical design*
- The concept of modularity is used in designing the system by defining some library files (header files).
- A *module* is a programming language entity that packages a set of code elements into a reusable unit.
- In object-oriented languages, the *class* is the primary unit of *modularization*.
- Modules allow us to decompose a complex system into more intellectually manageable pieces.

Hierarchy

Hierarchy is a ranking or ordering of abstractions

- A set of abstraction often forms a hierarchy, and by identifying such hierarchies in our design we greatly simplify our understanding of the problem.
- Two most important hierarchies in a complex system are
  - Class Structure (the "**is a"** hierarchy)
  - Object Structure (the "**part of**" hierarchy")

Examples of Hierarchy: **Inheritance**

- Inheritance is the most important "is a" hierarchy, and it is an essential element of object-oriented system.
- Inheritance defines a relationship among classes , wherein one class shares the structure or behavior defined one or more classes.

Inheritance Types:

- Single Inheritance : Only one superclass
- Multiple Inheritance: More than one superclasses

- Multilevel Inheritance : It is a level of hierarchy

Examples of Hierarchy**: Aggregation**

- is a" hierarchies denotes Inheritance (Generalization/specialization) relationships.
- "part of" hierarchies describes aggregation relationships.
- Aggregation is the process of forming a new complex aggregate object by joining smaller less complex classes together
- **Aggregation is the part-of relationship**

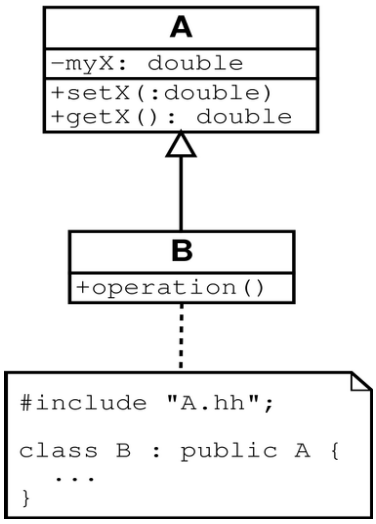


Fig: Inheritance

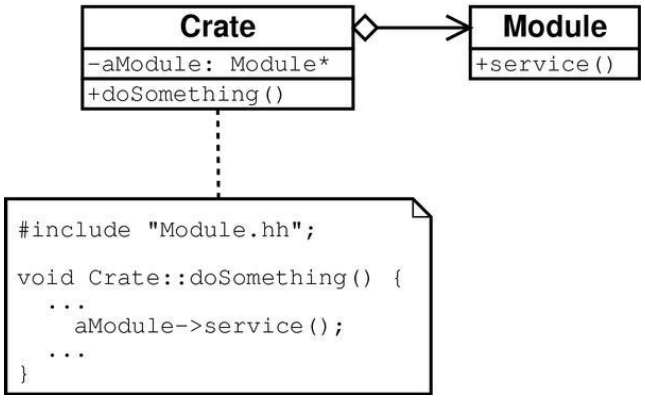


Fig: Aggregation

Typing

Typing is the enforcement of the class of an object, such that the objects of different types may not be interchanged, or at most they may be interchanged only in very restricted ways"

- The concept of type derives primarily from the theories of abstract data types.
- *A type is a precise characterization of structural or behavioral properties which a collection of entities all share.*
- The word type and class is used interchangeably.
- It closely related to the OOP language

Static and dynamic typing

- **Static or early binding** refers to the time when names are bound to types. Static binding means that the types of the variables and expression are fixed at time of compilation.
- In **Dynamic Binding** ,or late binding the types of all variables and expressions are not known until run time
- Strongly typing refers to type consistency.

Strongly Typed language

- A strongly typed language is one in which all expressions are guaranteed to be type consistent.

Concurrency

- For certain kind of problems, an automated system may have to handle many different events simultaneously

- Concurrency is the property that distinguishes an active object from one that is not active. Concurrency allows different objects to act at the same time.
- Heavyweight process: A heavyweight process is independently managed by target operating system and it has its own address space (and execution environment)
- Lightweight process: A lightweight process usually lives within a single operating system process along with other lightweight process, which share the same address space.

Persistence

- Persistence is the property of an object through which its existence transcends time (i.e. the object continue to exist after its creator ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created).
- Persistence saves the state and class of an object across time or space.
- Persistence objects are widely used in the object-oriented database system.

Benefits of Object Model

The Object Model offers a number of significant benefits that other models simply do not provide.

- Object model helps to us to exploit the expressive power of object-based and object-oriented programming
- Object model help to create the reusable application framework (i.e Reusability of software components)
- Object model leads more easy to manage changes
- Object model reduces the development risk and increases our confidence in the correctness of design
- The object model appeals to the working of human cognition.

Class and Objects

- An **object** has state, exhibits some well-defined behavior and has a unique identity.
- An object has **state** , **behavior**, and **identity**, the structure and behavior of similar objects are defined in their common class; the term instance and object are interchangeable
- The **state** of an object encompasses all of the properties of object plus current values of each of these properties.
- **Behavior** is how an object acts and reacts, in terms of its changes and message passing. In other word behavior of an object represents its outwardly visible and testable activity
- An **operation** denotes a service that a class offers to its client. In practice we have found that a client typically performs five kinds of operations up on an objects
  - **Modifier:** an operation that alters the state of an object (e.g. in Queue clear(), remove(), append() are modifiers)
  - **Selector:** an operation that accesses the state of an object, but does not alter the state.(e.g.length(),front(), isEmpty(), isFull() etc are selectors)

- **Iterator:** An operation that permits all parts of object to be accessed in some well-defined order.
- **Constructor**: An operation that creates objects and/ or initializes its state.
- **Destructor:** An operation that frees the state of an objects and/ or destroy the object itself

- **Active Object:** An active object is one that encompasses its own thread of control
  - Active objects are autonomous meaning that they can exhibit some behavior without being operated upon another object.
  - Sequential system usually have only one active object, In such system all other objects are passive
- **Passive Object:** An passive object does not encompass its own thread of control.
  - Passive object can undergo a state change when explicitly acted upon.
  - If the control tends to be distributed then there are passive objects (e.g. transaction processing system)

- We have said that every object is an instance of some class.
- A **Metaclass** is a class whose instances are themselves classes. Language such as Smalltalk supports concept of metaclass ,C++ does not support it.

Unified Modeling Language

- Primary goal of SDLC is to produce the software that meets all the requirements.
- To develop the software of lasting quality, Modeling works as an strong architectural foundation as well as the central part of activities that leads up to the development and deployments of a good software.
- It provides clear understanding of desired structure and behavior of the system.
- Architecture, process and Tools are issues of quality software engineering.
- In addressing these issues modeling is proven and well accepted engineering technique.

Four aims of a model:

- Help us to visualize a system as it is or as we want.
- Permits us to specify structure and behavior of a system.
- Give us the template that guides us in constructing a system.
- Document the decision we have made.

UML: An overview

- Booch, Rumbaugh and Jacobson invented a standard language for software design called UML.
- UML is a language that addresses different views of a system's architecture as it evolves throughout the SDLC.
- The vocabulary and rules of UML tells us how to create and read well-formed models.
- UML. In broader sense, is language for visualizing, specifying, constructing and documenting the artifacts of OO software system?

Building blocks of UML

There are three kind of building blocks:

- Things: First level of abstraction in UML.
- Relationships: Tie things together.
- Diagrams: Group Collection of Things.

There are four types of things:-

- Structural things
- Behavioral things
- Grouping things
- Annotational Things

Structural Things:

- They are nouns of model and are static parts of a model.
- They represent either physical or logical things of a model.

- There are 7 different structural things:

  - Class
  - Interface
  - Collaboration
  - Use Case
  - Active Class
  - Component
  - Node

Behavioral Things:

- They are dynamic parts of UML models- generally verbs of model, representing behavior over time and space.
- There are primarily two kinds of behavioral things:
  - Interaction:

    Consist of set of messages exchanged among a set of objects
  - State machine

    Specifies the sequences of states an object or interaction goes through.

Grouping Things:

Package is way grouping things in UML.

Annotational Things:

These are expananatory parts of UML.

Relationships in UML

Relationships join things together showing the kind of link a thing has with another thing.

- Dependency
- Association
- Generalization/Specialization
- Realization

Diagrams in UML:

A diagram is a graphical representation of a set of elements. More often a connected graph of things and relationships. There are 9 diagrams.

- Class Diagram
- Object diagram
- Use Case Diagram
- Sequence diagram
- Collaboration diagram
- Statechart diagram
- Activity diagram

- Component diagram
- Deployment diagram

Class Diagrams
Class diagrams are the backbone of almost every object oriented method, including UML. They describe the static structure of a system.

Object Diagrams
Object diagrams describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy.

Use Case Diagrams
Use case diagrams model the functionality of system using actors and use cases.

Sequence Diagrams
Sequence diagrams describes interactions among classes in terms of an exchange of messages over time. Gives dynamic view.

Collaboration Diagrams
Collaboration diagrams represent interactions between objects as a series of sequenced messages. Collaboration diagrams describe both the static structure and the dynamic behavior of a system.

Statechart Diagrams
Statechart diagrams describe the dynamic behavior of a system in response to external stimuli. Statechart diagrams are especially useful in modeling reactive objects whose states are triggered by specific events.

Activity Diagrams
Activity diagrams illustrate the dynamic nature of a system by modeling the flow of control from activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Typically, activity diagrams are used to model workflow or business processes and internal operation.

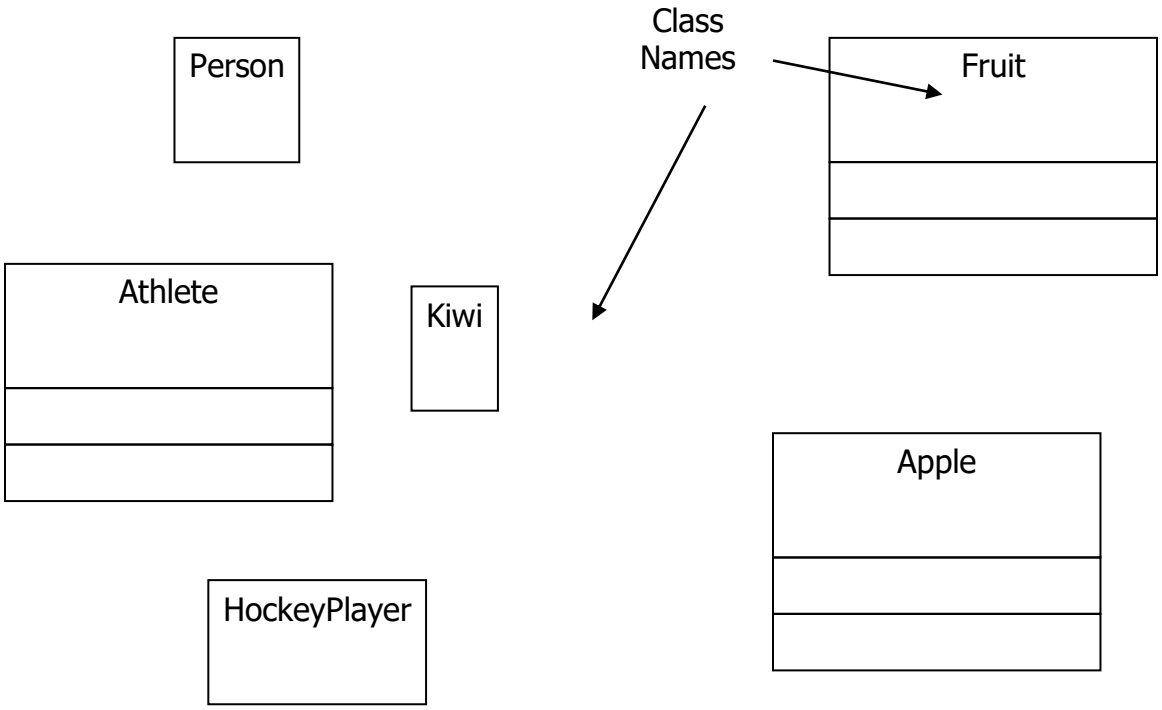
Component Diagrams
Component diagrams describe the organization of physical software components, including source code, run-time (binary) code, and executables.

Deployment Diagrams
Deployment diagrams depict the physical resources in a system, including nodes, components, and connections.

Class Diagrams
Depicting Classes and Static Relationships

- Class diagrams allow us to express classes in UML including attributes and methods
- Class diagrams allow us to declare our classes (visually)

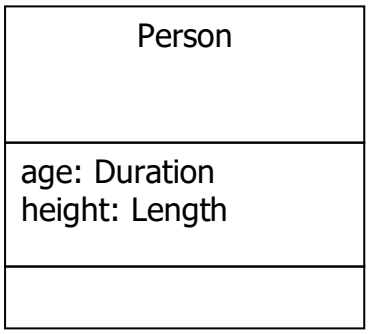
Classes in UML

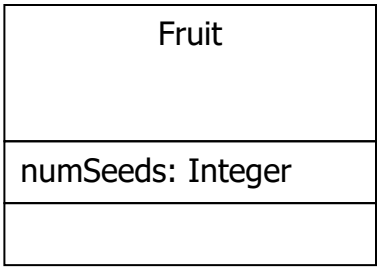


- As you can imagine, the class shapes shown here are a major part of class diagrams
  - Both forms are acceptable for classes that have no attributes or methods
  - The two empty blocks are used for holding declarations of attributes and methods
  - The top block is used to hold the class name

Attributes in Classes

- Attributes are specified using the form:
  - name : type
  - This format is taken from Eiffel, which uses a Pascal-like syntax
  - Names are usually alphanumeric
  - Types are logical types from the domain
    - In this example, an apple's diameter is a quantity of type 'Length', which could be represented as a float or double in C++ or Java



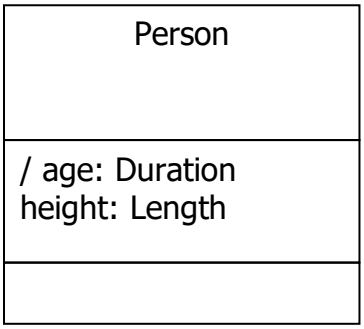


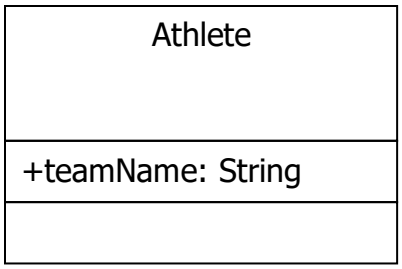
Attribute Modifiers

- Attributes can be read/write (default) or read only
  - Attributes that are read only are preceded with the modifier '/'
- Attributes can be defined on the objects (default) or the class
  - Class attributes are preceded with the modifier '$'

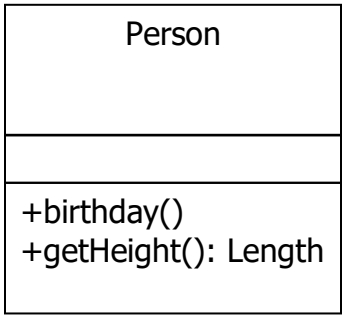
Attributes can have different visibilities

- Public attributes, denoted with the modifier '+', can be accessed from outside the class
- Private attributes, denoted with the modifier '-', can only be accessed from within the class
- Protected attributes, denoted with the modifier '#', can be accessed from within the class, or any descendant (e.g. a subclass)

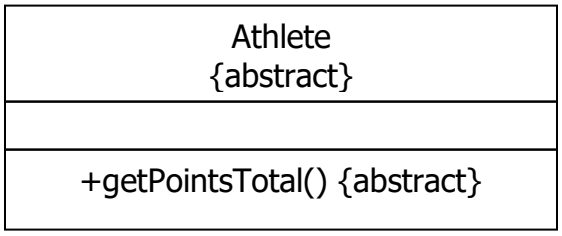




Operations in Classes



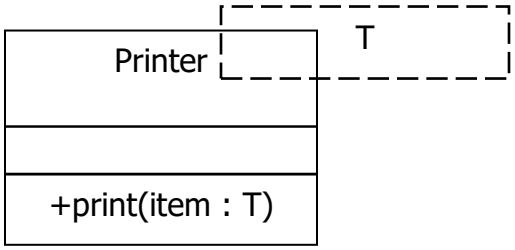
Abstract Classes and Methods

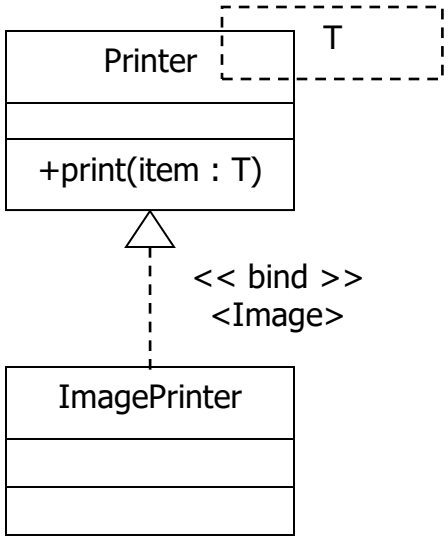


- The class Athlete was declared abstract
  - This is analogous to abstract classes in Java
- The getPointTotals() method was declared abstract
  - This means the method is not implemented in this class, but left for a (concrete) subclass to implement
  - This method being abstract makes it necessary for Athlete to be declared an abstract class

Genericity in Classes

- Genericity, or run-time determined types, in a class can be represented in UML
  - This allows you to design classes to work on a group of types, where the type is 'generic' at compile time
- The type, specified at run-time, is an argument or parameter to the class
  - Such classes are also called parameterized classes

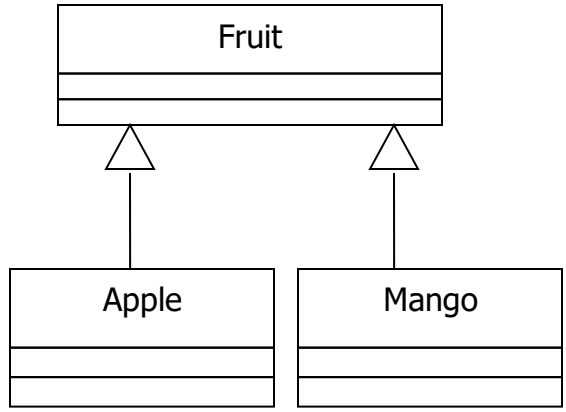


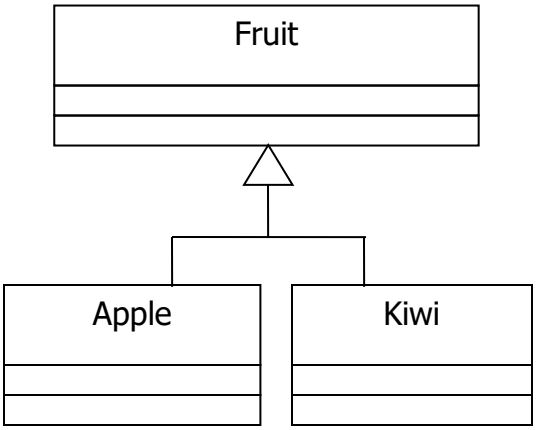


Relationships in a class

- There are three types of relationships:
  - A 'is a type of' B (inheritance)
  - A 'is associated with' B (association)
  - A 'is a part of' B (composition)

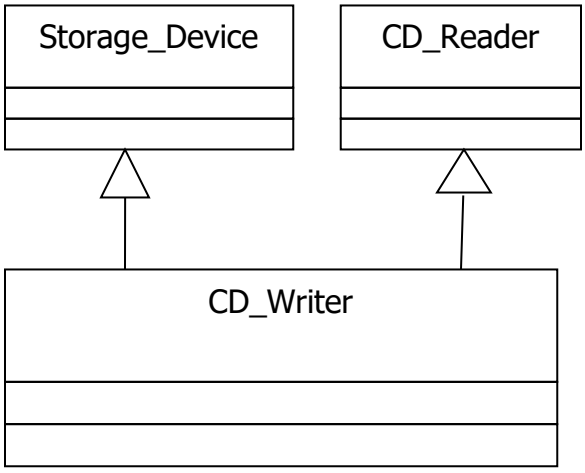
Inheritance





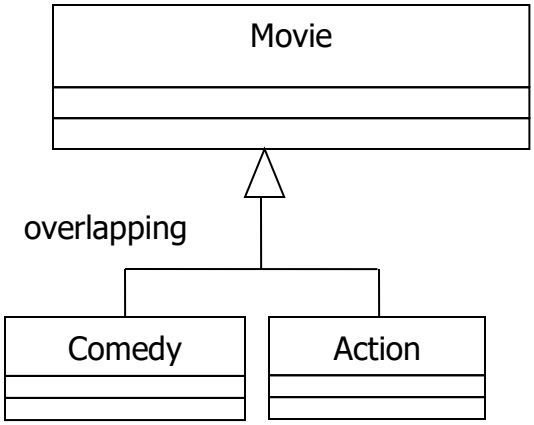
This is an alternate notation

Multiple Inheritance

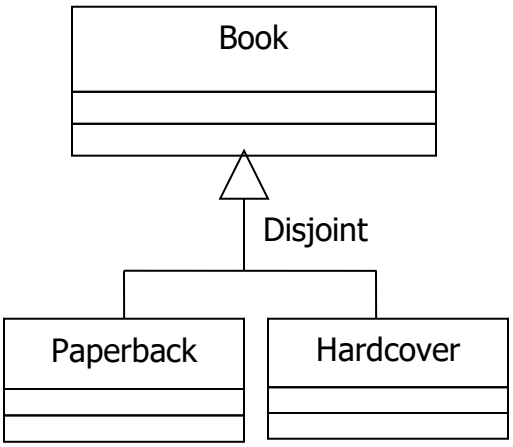


Groups

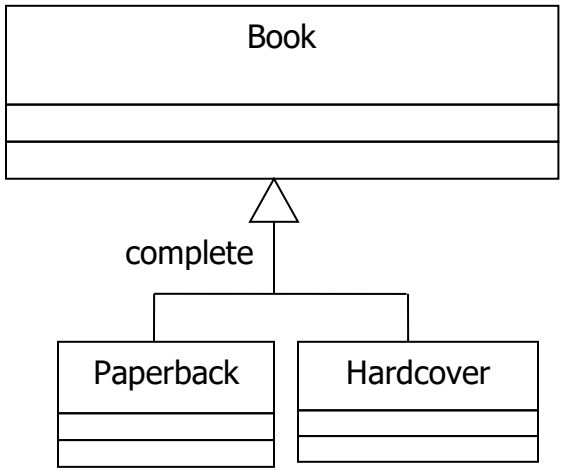
- Whenever we create a subclass, we are defining a new group of objects, that is a subset of the superclass' group
- To be even more complete, when we define inheritance, we usually use keywords to give details about those groups
- **Overlapping/Disjoint**
    - Overlapping: Groups are overlapping if there are objects that are members of both groups
    - Disjoint: Groups are disjoint if there are no objects that are members of both groups
    - Example: Say we have a superclass called 'Movie'
    - Say we have two subclasses of 'Movie' called 'Comedy' and 'Action'
    - Since it is possible for a movie to have both action and comedy, the two groups are overlapping



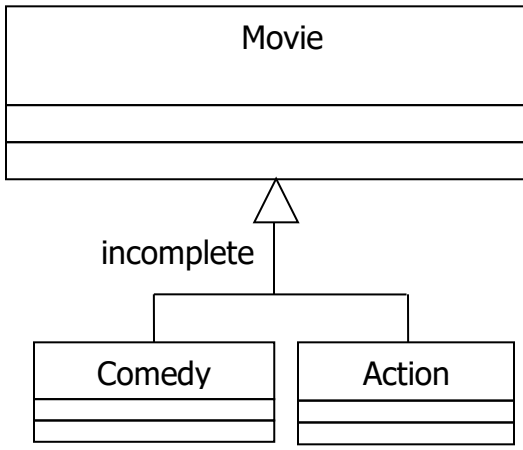
- Example: Say we have a superclass called 'Book'
- Say we have two subclasses of 'Book' called 'Paperback' and 'Hardcover'
- Since a book is either a paperback or a hardcover book, the two groups are disjoint



- **Complete/Incomplete**
  - **Complete:** The groups listed contain all possible objects that belong to the superclass' group
    - i.e. No more disjoint subclasses can be defined
  - **Incomplete:** The groups listed do not contain all possible objects that belong to the superclass' group
  - Example: Consider our superclass 'Book' with its subclasses 'Hardcover' and 'Paperback'
  - Are there any other kinds of book?
    - The answer is 'no', this the groups are complete



  - Example: Consider our superclass 'Movie' with its subclasses 'Comedy' and 'Action'
  - Are there any other kinds of movie?
    - The answer is 'yes', this the groups are incomplete



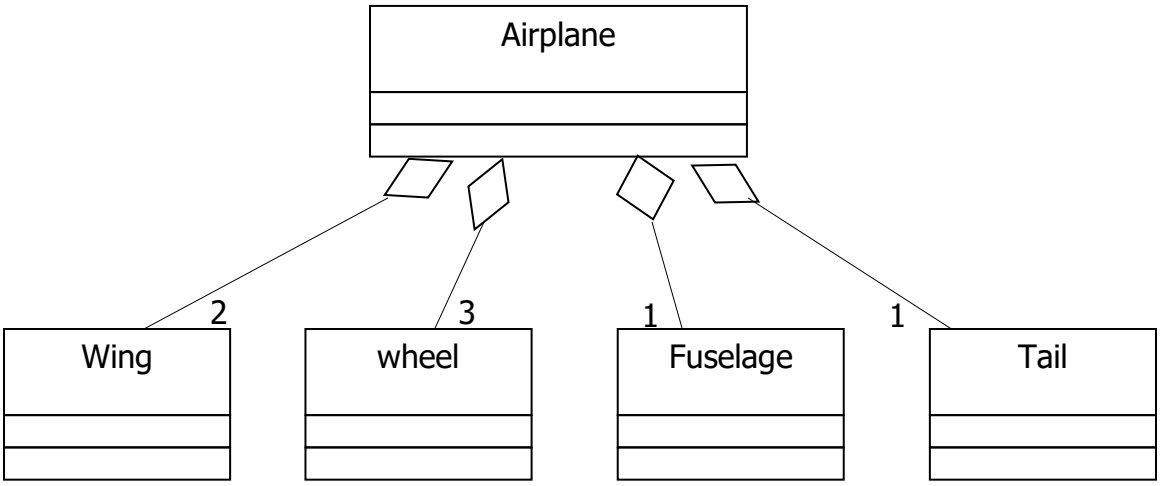
Association

- Associations between classes represent logical relationships objects of those classes might hold
- For example, a 'Person' class might have an association 'HomeAddress' with another class 'Location'

  - This is because an instance of Person might have an instance of Location as its home address
- An association contains:
  - The name of the association
    - This name describes the association
  - Multiplicity at each end of the association
    - This describes the number of instances that may occur at that end of the association
  - The role of the class at each end (optional)
    - This is usually a name used to describe the instances with respect to that association
- Let's illustrate these concepts with an example
  - Say we have two classes, 'Person' and 'MusicalInstrument' , and an association called 'Plays'
  - The association Plays represents all instances where a Person plays a Musical Instrument

Composition

- Composition is a relationship between two classes where one class is a part of another (an 'is a part of' relationship)
  - If a class A is composed of classes B, C, and D, we may say that A has components B, C, and D
  - Compositions typically consist of a name, as well as the multiplicity of the component
- If a class 'Car' has a composition relationship with another class 'Wheel', we should agree that normally the multiplicity of Wheel is 4 (exactly)
- 'Vehicle', on the other hand, could have any number of wheels (0..*)
- Composition is a special form of association
  - It is important enough to have its own notation



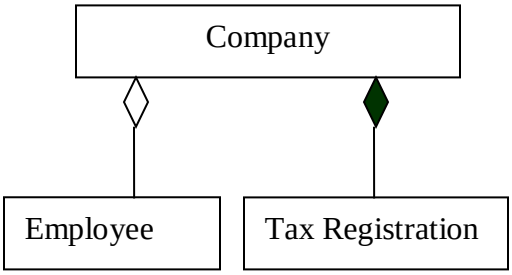
Aggregation

- Aggregations is also a type of association, but again, a special one that is given its own notation
  - This is because, like composition, it is very common
- An aggregation is another relationship between classes which says one class 'is a part of' another class
- Both aggregation and composition represent the 'is a part of' relationship
- The main difference is what the part represents
  - In aggregation, the part (constituent) is meaningful without the whole (aggregate)
  - In composition, the part (component) is not meaningful without the whole (container)

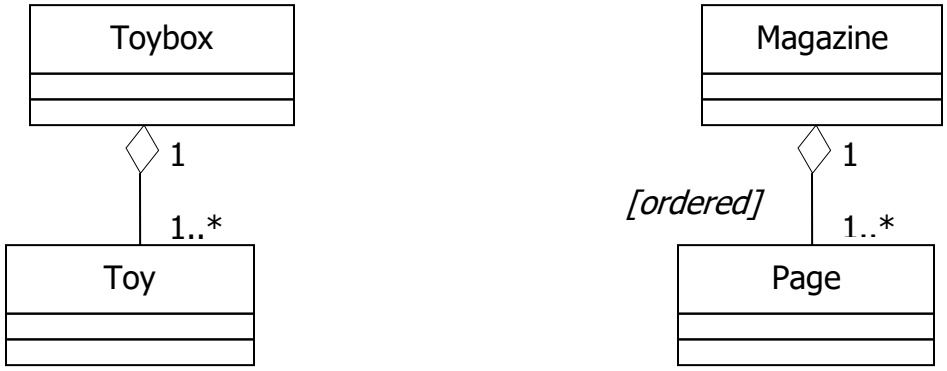
Example:

Aggregation: A company has employees. Employees may change company. Employees exist irrespective to companies.

Company: A company has a tax registration. The tax registration is tied with the company and dies with it.

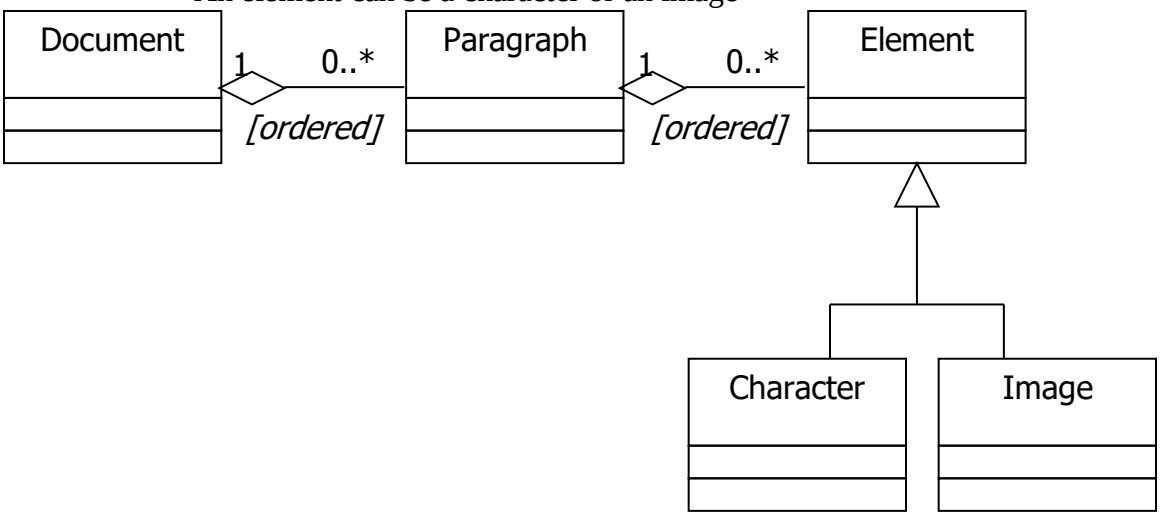


- In UML, composition is considered a special case of aggregation where constituents belong only to a single aggregate
  - This is why some tools (such as Rational Rose) only have symbols for aggregation, and not composition
- Aggregates can be ordered or unordered
  - In ordered aggregates, the order of the constituents within the aggregate is important
  - In unordered aggregates, the order is unimportant
- Ordered aggregates are indicated using the symbol *[ordered]*
  - Unordered is the default for aggregates



Class Diagrams: An Example

- Say we are writing a word processor
  - A word processing document consists of a number of paragraphs
  - Each paragraph consists of a number of elements
  - An element can be a character or an image

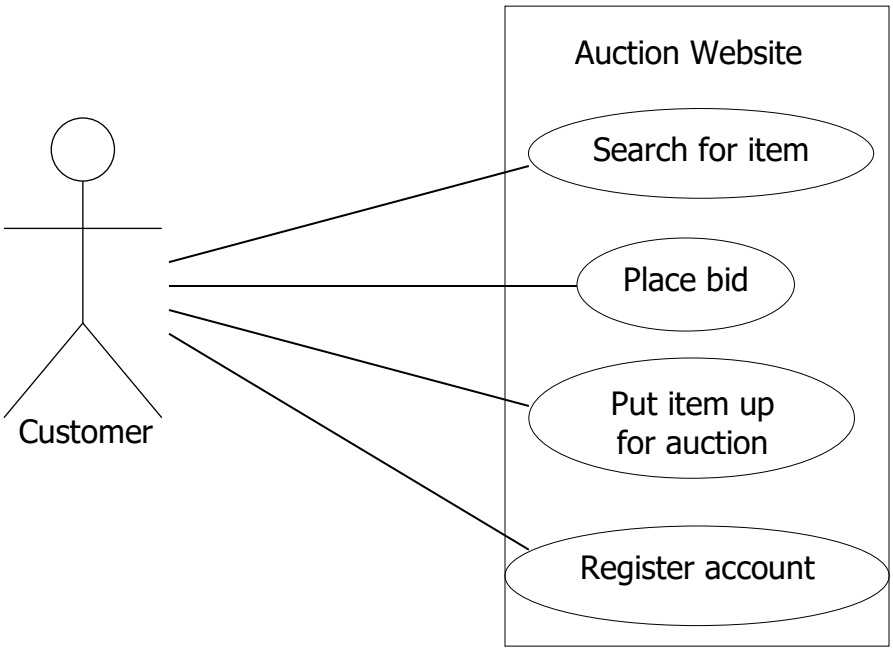


Use Case Diagrams

- Use case diagrams describe relationships between users and use cases
- A use case is a (usually high-level) user activity in the system
  - Use cases represent system functionality, the requirements of the system from the user's perspective.
  - A use case is a collection of related success and failure scenarios that describe actors using a system to support a goal
  - A use case name should always begin with a verb
- Use case diagrams consist of at least two components:
  - **An actor**
    - Actors are represented as stick people, with a label below, naming the actor's role
    - There may be multiple actors in a diagram
  - **A use case**
    - Use cases are represented as ellipses, with a label inside, naming the use case
    - There may be multiple use cases in a diagram

Use Cases

- For an example, consider an auction website (e.g. eBay)
- The actor *customer* might:
  - Register for an account
  - Search for items by keyword
  - Place a bid on an item
  - Put an item up for auction
  - Check the status of this item



Use Case Diagrams

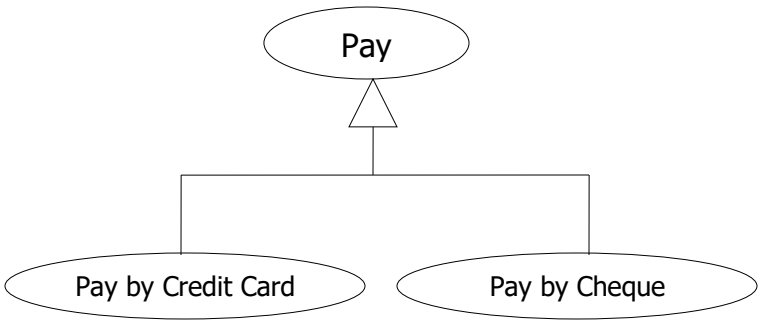
- Obviously, actor and use case names must be unique within a given diagram
- Consider use cases to be similar to classes:
  - A use case describes an activity that is possible
  - A given system may have several instances of that activity throughout its lifetime

Use Case Inheritance

- Consider the following use case from the auction website example:
  - Place bid
- Say that it is possible for a customer to pay for a purchased item by sending a cheque to the auctioning customer
- What if it is also possible for a customer to pay using his/her credit card?
- This is an example of use case inheritance
  - A use case 'Pay with credit card' might inherit from the use case 'Pay'
  - In UML (and OOAD) terminology:
    - 'Pay with credit card' is a specialization of 'Pay'
    - 'Pay' is a generalization of 'Pay with

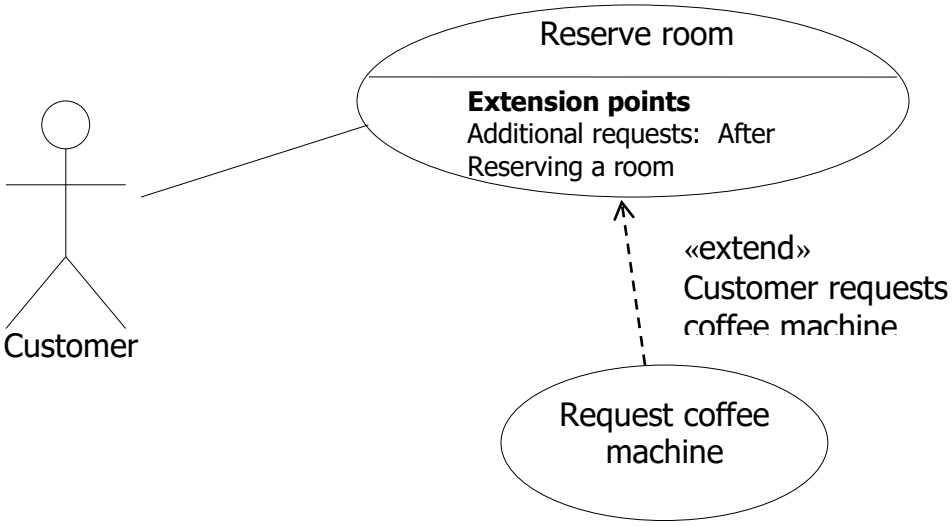
e.g. pay(card: CreditCard), pay(cheque: Cheque)

- Use case inheritance is shown with a solid arrow, with a triangular arrowhead
  - The arrow points from the *specialized* use case to the *generalized* use case



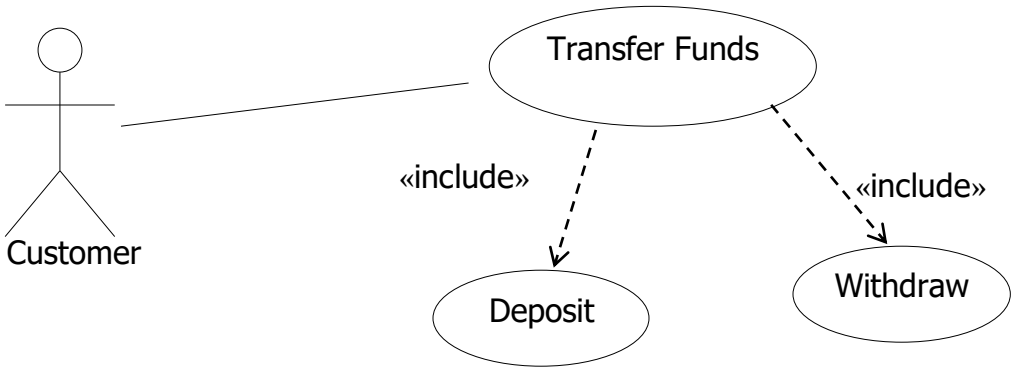
Use Case Extensions

- Use cases can be extended by other use cases
- Usually, conditions exist somewhere in the use case's activity that would cause the extended use case to occur
  - This 'somewhere' is called an *extension point* in UML
- The original use case is called the *base use case and t*he new use case is called the *extended use case*
- The extended use case usually augments the base use case with additional behaviour
- For example, a use case for a hotel might be 'reserve hotel room'
- An extension to this use case might be 'request coffee machine' to request that a coffee maker be in the room
- The customer still wants to reserve a hotel room, but they also want a coffee maker in their room when they arrive
- In UML use cases extensions are specified using a dashed arrow
  - The arrow points from the extended use case to the base use case
  - The arrows are labelled with «extend» plus a description of the conditions which cause the extension to occur
  - The base use case usually contains a description of the extension points where the use case may be extended



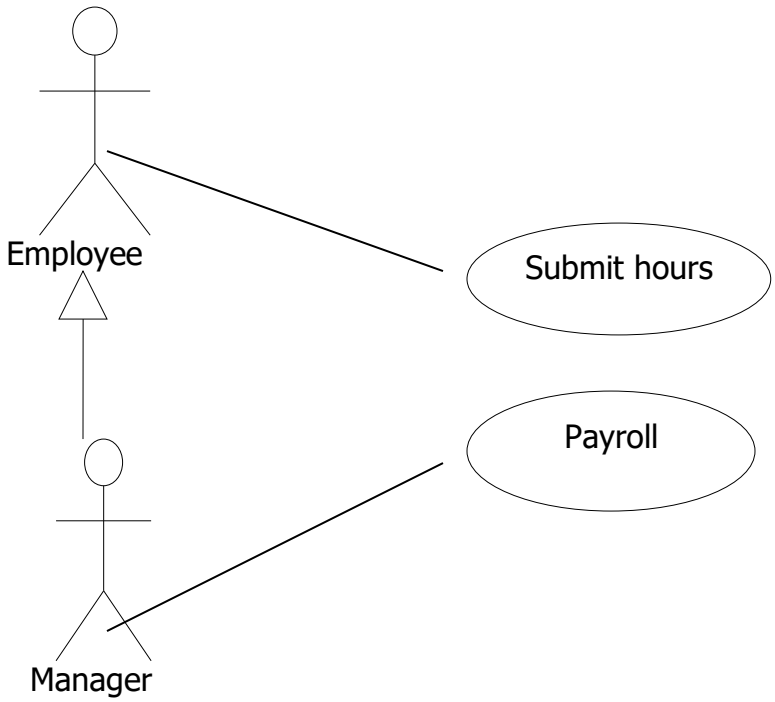
Use Case Inclusion

- Use case inclusion is when one use case is used within another use case
  - For example, the use cases 'withdraw' and 'deposit' in a bank, might be used within the use case 'transfer funds'
- Use case inclusion is specified using a dashed arrow
  - The arrow points from the container use case to the included use case. The arrow is labelled with «include»

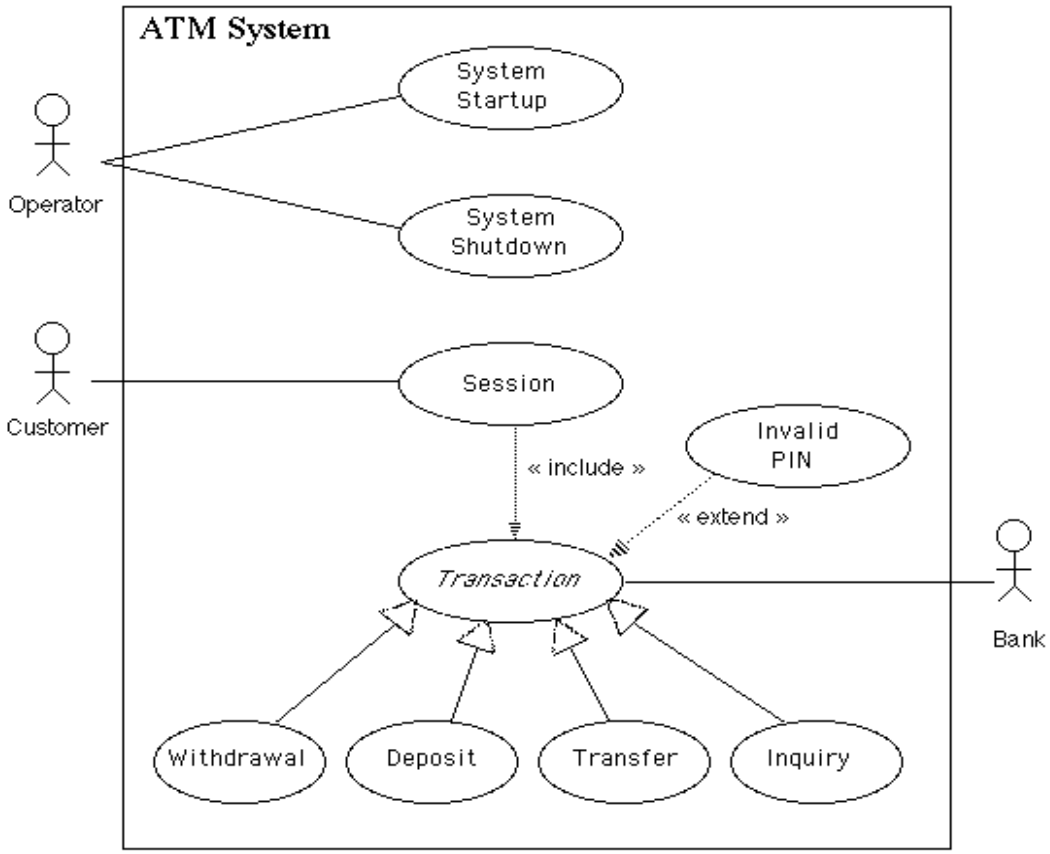


Actor Inheritance

- It may also make sense to have actor inheritance in a system
- For example, an employee might have a specialization called manager
  - A manager might handle (in addition to normal employee duties) customer complaints, refunds, scheduling, etc.



Use case Diagram Example

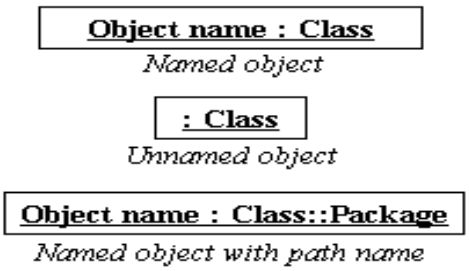


UML Object Diagram

- Object diagrams are also closely linked to class diagrams. Just as an object is an instance of a class, an object diagram could be viewed as an instance of a class diagram.
- Object diagrams describe the static structure of a system at a particular time and they are used to test the accuracy of class diagrams

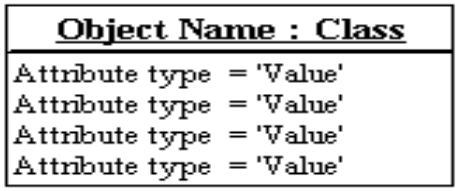
Object names

- Each object is represented as a rectangle, which contains the name of the object and its class underlined and separated by a colon.



Object attributes

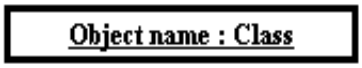
- As with classes, you can list object attributes in a separate compartment. However, unlike classes, object attributes must have values assigned to them.



Object with attributes

Active object

- Objects that control action flow are called active objects. Illustrate these objects with a thicker border



Active object

Object-Interaction Diagrams

Defining Behaviour in UML

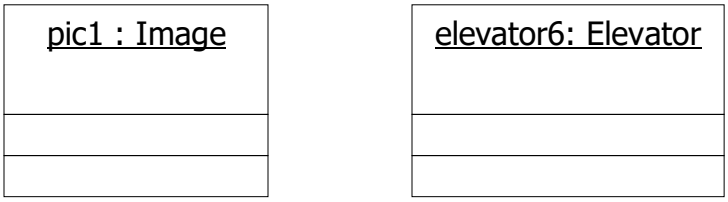
- Class diagrams represent the static relationships between classes
- In contrast, object-interaction diagrams represent dynamic relationships between objects
- Often, there are numerous relationships that exist between objects
  - Thus, for simplicity, object relationships are usually defined using separate diagrams for each 'action unit' called a *use case*
- There are two types of object-interaction diagrams:
  - A collaboration diagram
  - A sequence diagram
- Each of these diagrams can be used interchangeably

Collaboration Diagrams

Depicting Objects

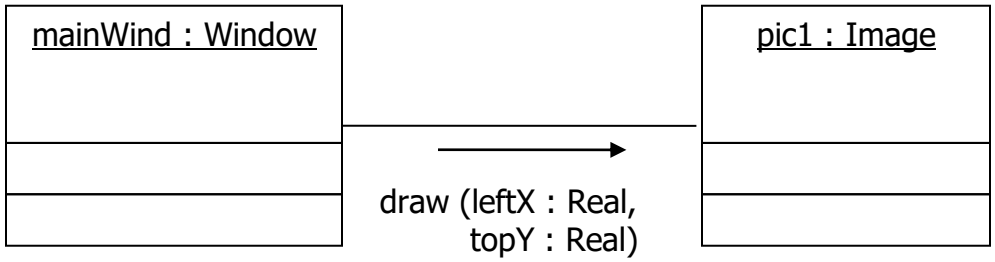
- Objects are defined using a similar notation to classes
  - In addition to the class name in the top box, the object instance name is also specified

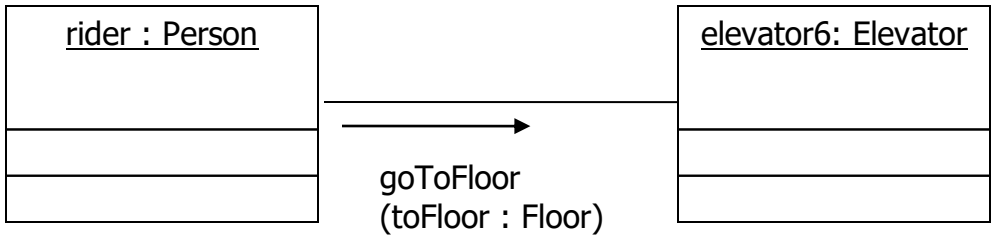
– The entire class declaration (class and instance name) is underlined to indicate it is an instance



Depicting Messages

- Messages are depicted with communication paths (lines)
  - The direction of the message is indicated with an arrow
  - The name of the operation (method name) is given, along with the names and types of any arguments
- e.g. Consider the following messages:
  - pic1.draw(x, y);
  - elevator6.request(3);

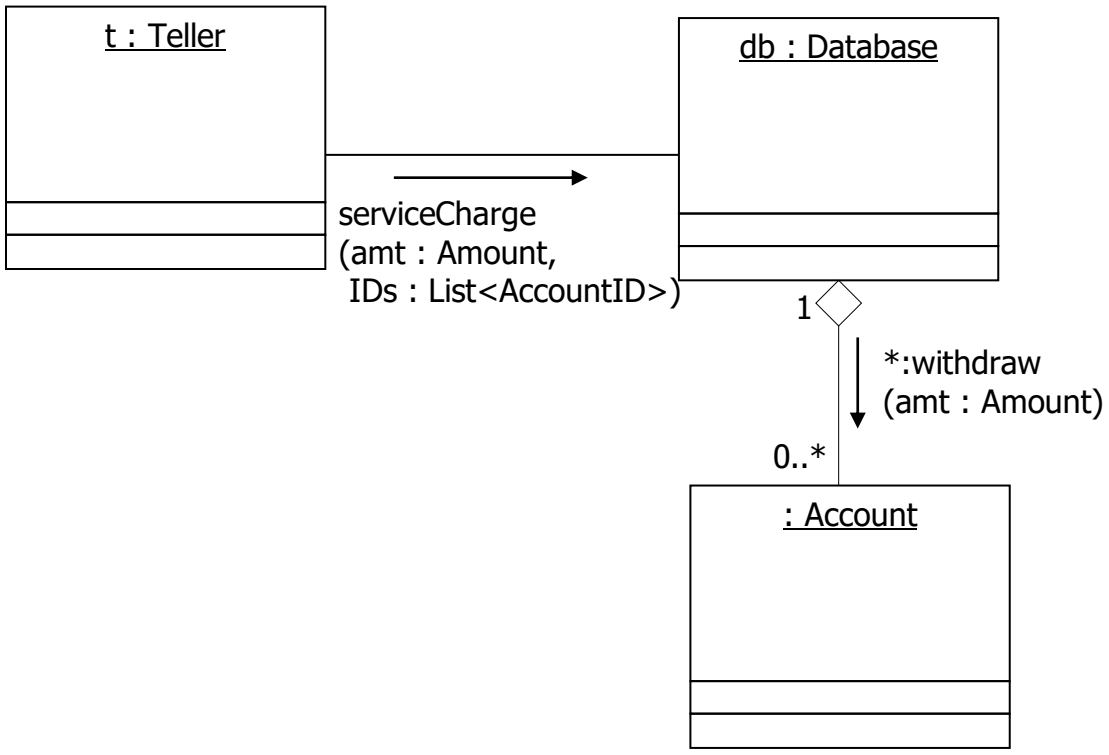




Message Multiplicity

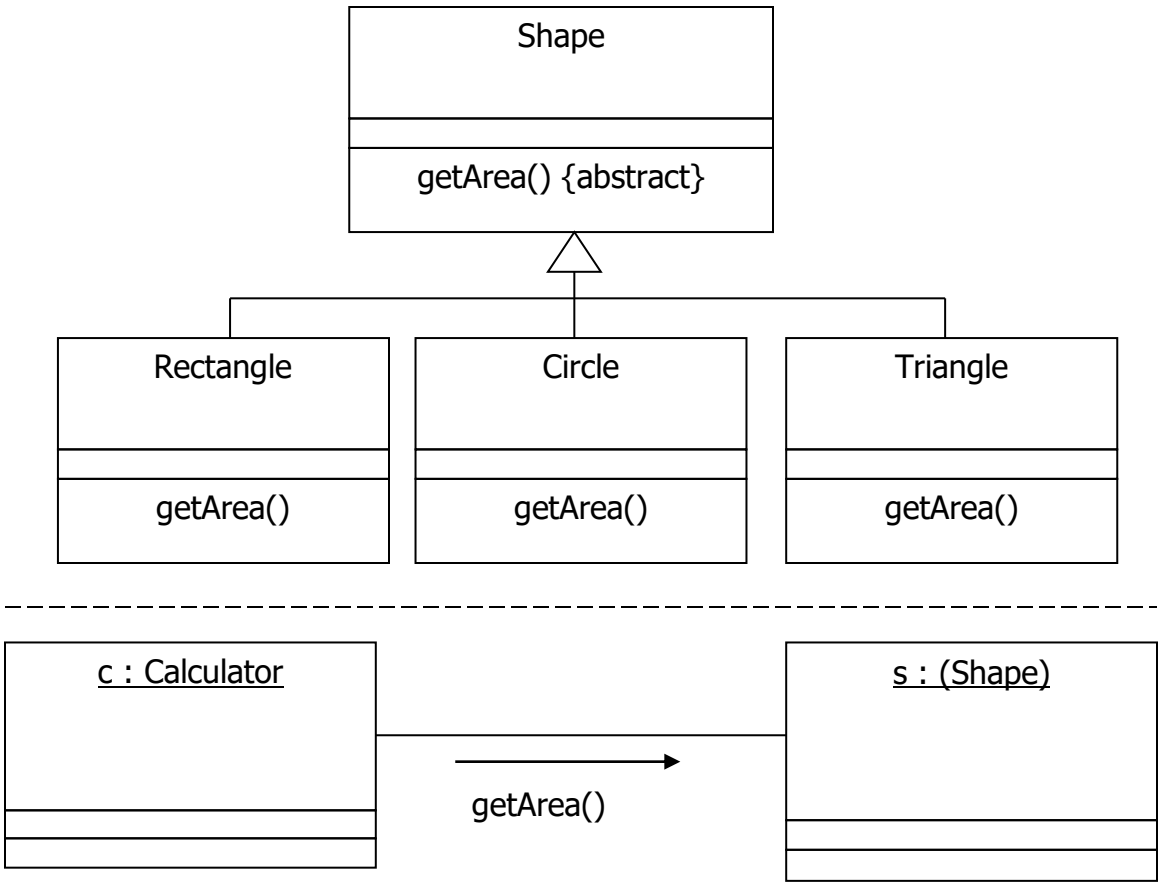
- Message multiplicity is particularly useful when an class has an aggregate relationship with another class
  - Message multiplicity can be used to denote that an aggregate object might repeatedly call the same operation on each of its constituents

- We can assume that 'Database' is an aggregate with a number of 'Account's as its constituents
- The 'serviceCharge' operation on 'Database' would involve calling 'withdraw' on all of its 'Account' constituents



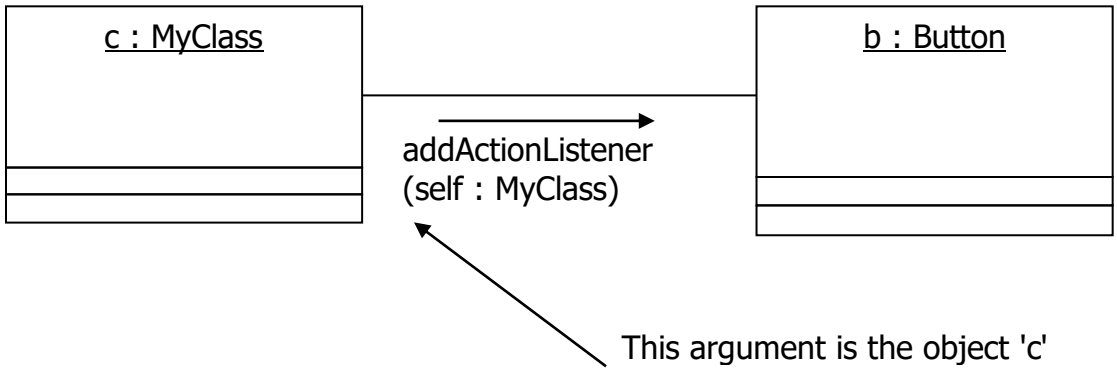
Message Polymorphism

- As you are aware, messages which are calls to operations on objects, may involve polymorphism
  - If a message may be polymorphic, you can represent this in UML by displaying the class name in brackets, '(' and ')'



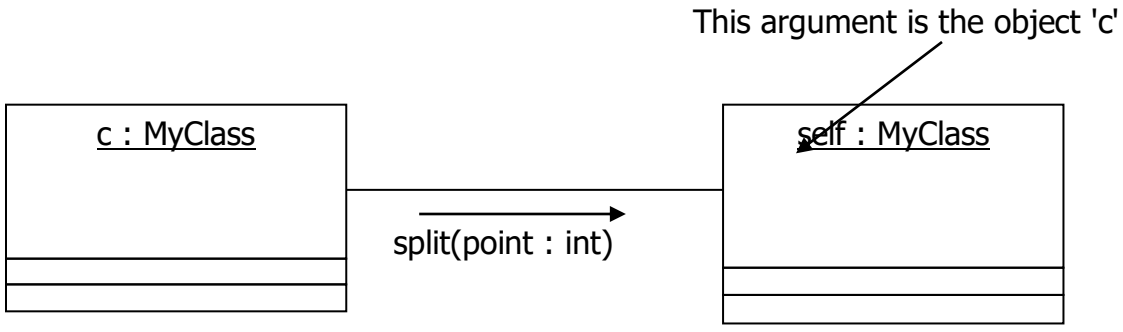
Self-Referencing Messages

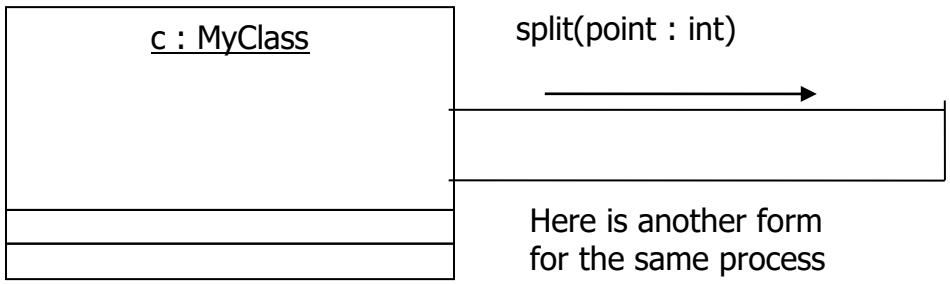
- Sometimes, it is relevant for a message to contain a reference to the source object
  - For example, say we are creating a class that generates some kind of event(s). Several objects may indicate to instances of that class that they should be notified when events occur
  - This type of thing occurs commonly in Java (although in Java it is called 'this', not 'self'): button.addActionListener(this);
- In situations like these, if an argument to an operation is the source object itself, it is always named 'self'



Self-Addressed Messages

- In other times, a message is sent from an object to itself
  - For example, a 'quicksort' operation on a class called 'List' might call an operation called 'split' on itself
- In situations like this, the destination object is always named 'self'



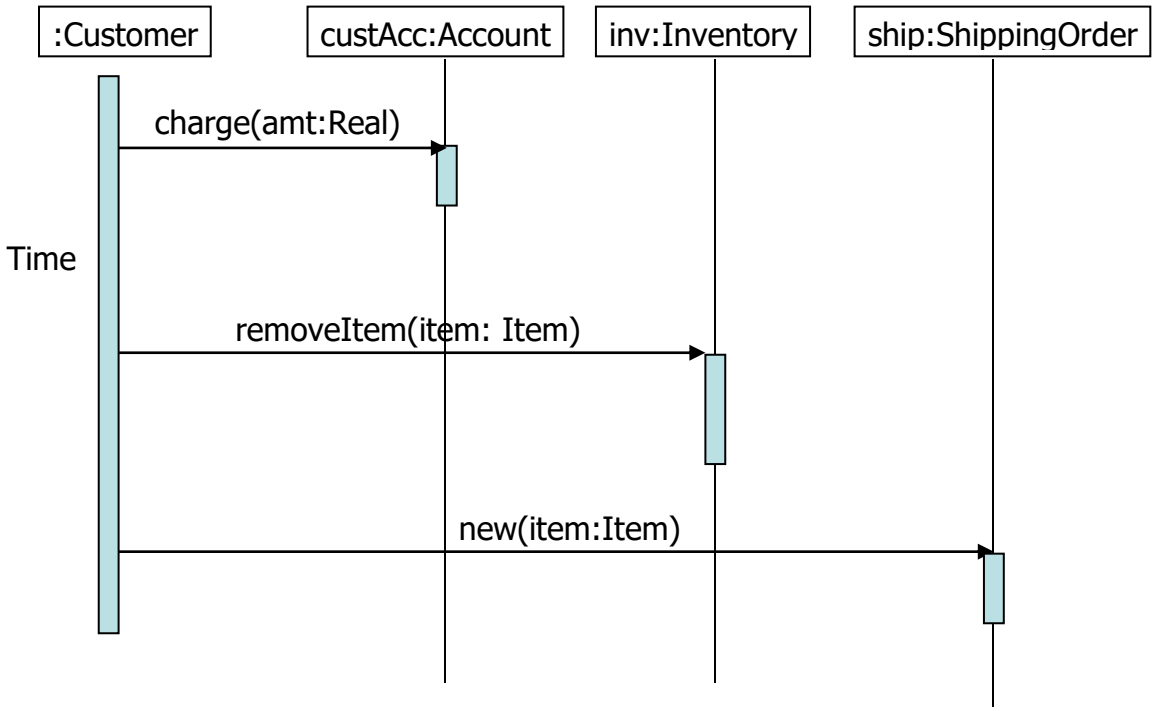


Sequence Diagrams

- Collaboration diagrams are used to show simple interactions between objects
  - While it is possible to define complicated behaviour using collaborations, often the diagrams become very visually complicated and difficult to follow
  - Sequence diagrams are specifically suited to defining complex interactions over a period of time
- It is important to remind you that collaboration and sequence diagrams can be used interchangeably
  - However, each have their specific uses
- Sequence diagrams show interactions between objects over time
- Sequence diagrams are often preferable to collaboration diagrams when:
  - There are several interactions involved in a particular behaviour
  - The sequence (temporal placement) of these interactions gets fairly complicated
- Most of the symbols in sequence diagrams are identical to those used in collaboration diagrams
  - The biggest differences lie mainly in the layout of these symbols into a complete diagram
  - Sequence diagrams are similar to graphs, where the y-axis represents the passage of time

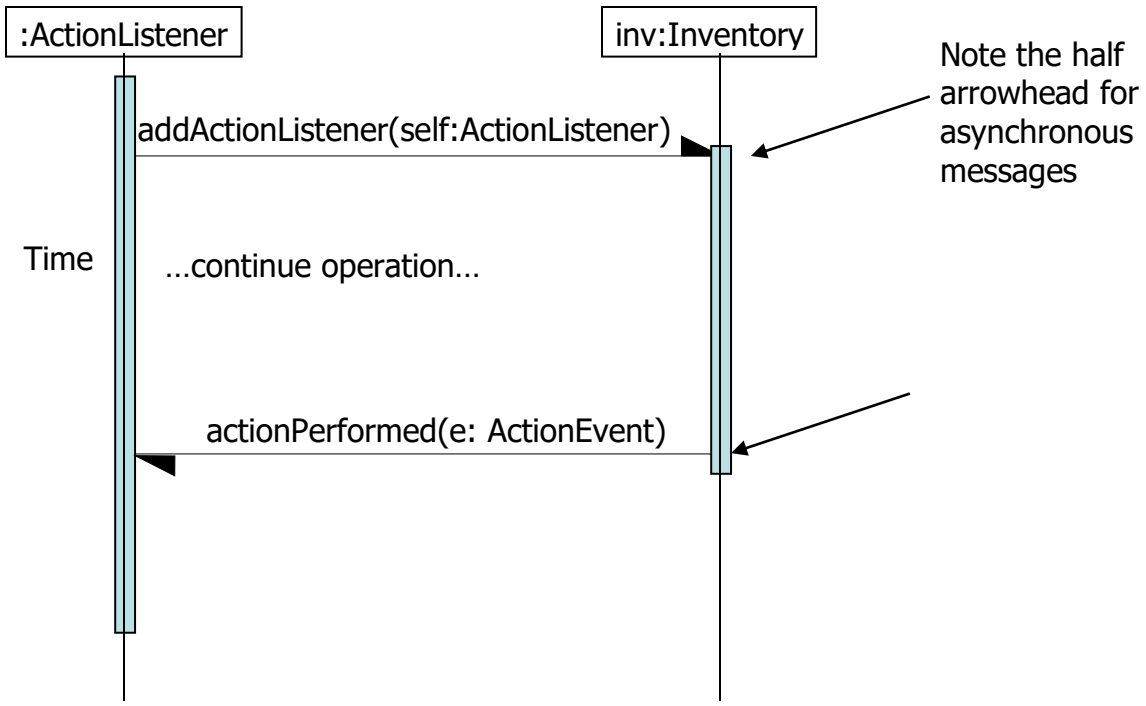
Sequence Diagrams: Example

- Consider an operation called purchase in a warehouse environment
- It may involve a number of steps:
  - Charge the price of the item (plus tax) to the customer's account
  - Removing the product from the inventory database
  - Creating a shipping order for the product when it arrives so the item will be delivered to the customer's address



Asynchronous Messages

- The messages discussed so far represent synchronous messages
  - The sender of the message waits for the receiver to complete the task (operation) before it continues
- Another type of message is possible: asynchronous
  - The sender of the message continues its own operation immediately after issuing the message
  - Eventually, a response may be received from the receiver
- An example of this is a Timer in Java:
  - An object that implements ActionListener may register itself as a listener for a Timer object
  - The Timer object will store a handle to the object, but then return control to the object after it does so
  - Later, when the Timer's duration has expired, an event will be issued to the object's 'actionPerformed' method
- Obviously, if the object had to wait for the timer to expire, it could have missed a long period of potential computation time



- Both the message 'addActionListener' and the callback 'actionPerformed' in our example are asynchronous messages
- Of course, more than one callback is possible as a response to the original request
  - This is true in general, as well as in our example

Message Priority

- With asynchronous messages, there is a possibility of messages being received faster than they can be processed
  - When this occurs, the messages are normally stored in a queue, and processed in the order in which they arrived
  - UML provides notation that allows this to be extended to provide priority
    - Instead of a true queue, higher priority messages can be handled with less delay than lower priority messages
    - This uses a form of prioritized queue
- Priority can be depicted by adding a tag '{priority = 3}' to the message name
  - E.g. actionPerformed(e:ActionEvent)
    {priority = 3}

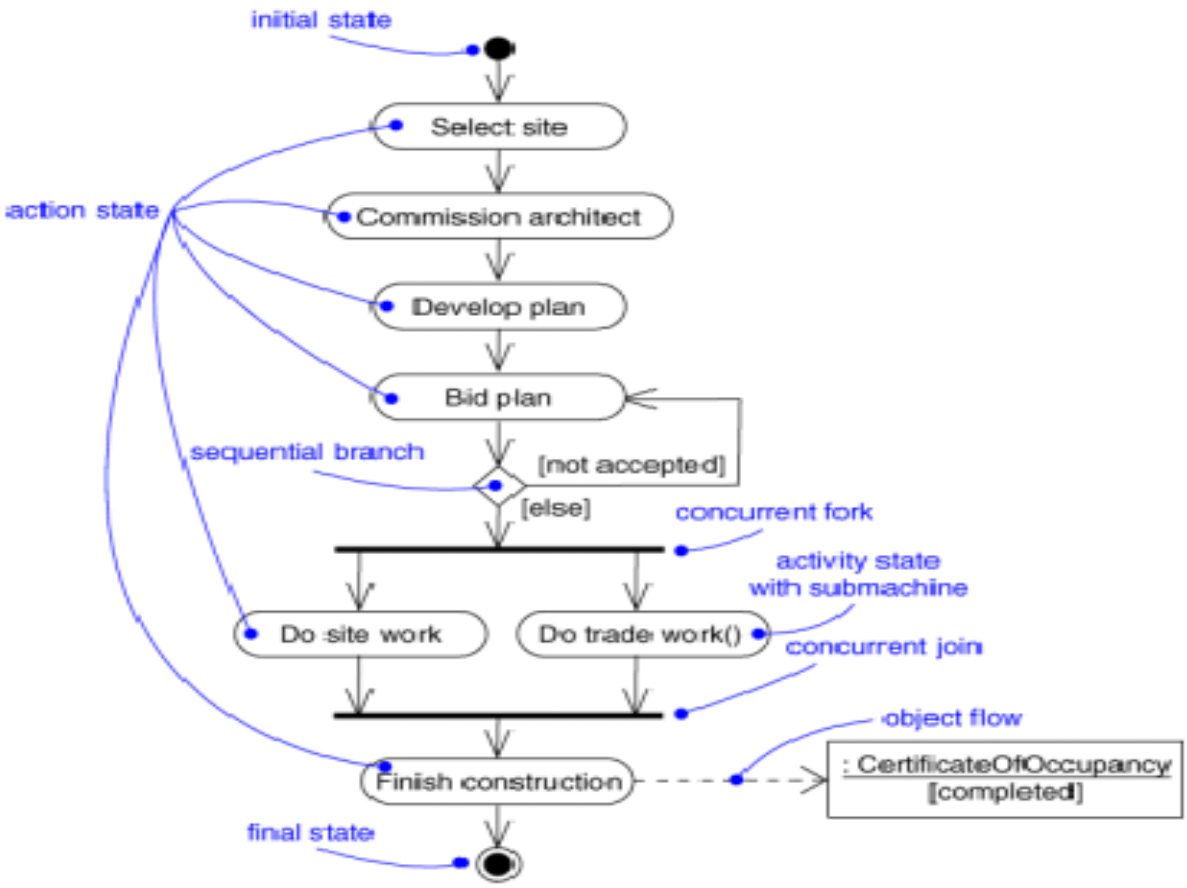
Activity Diagrams

- An activity diagram shows the flow from activity to activity
- An activity diagrams are one of the five diagrams in UML for modeling the dynamic aspects of system that shows flow of control from activity to activity.
- Activity diagrams show the flow of activities through the system
- Activity diagram captures dynamic behavior (activity-oriented)
- Behavior that occurs within the state is called an **activity**: starts when the state is entered and either completes or is interrupted by an outgoing transition.

- Common use of Activity Diagram
  - To model a workflow
  - To model an Operation

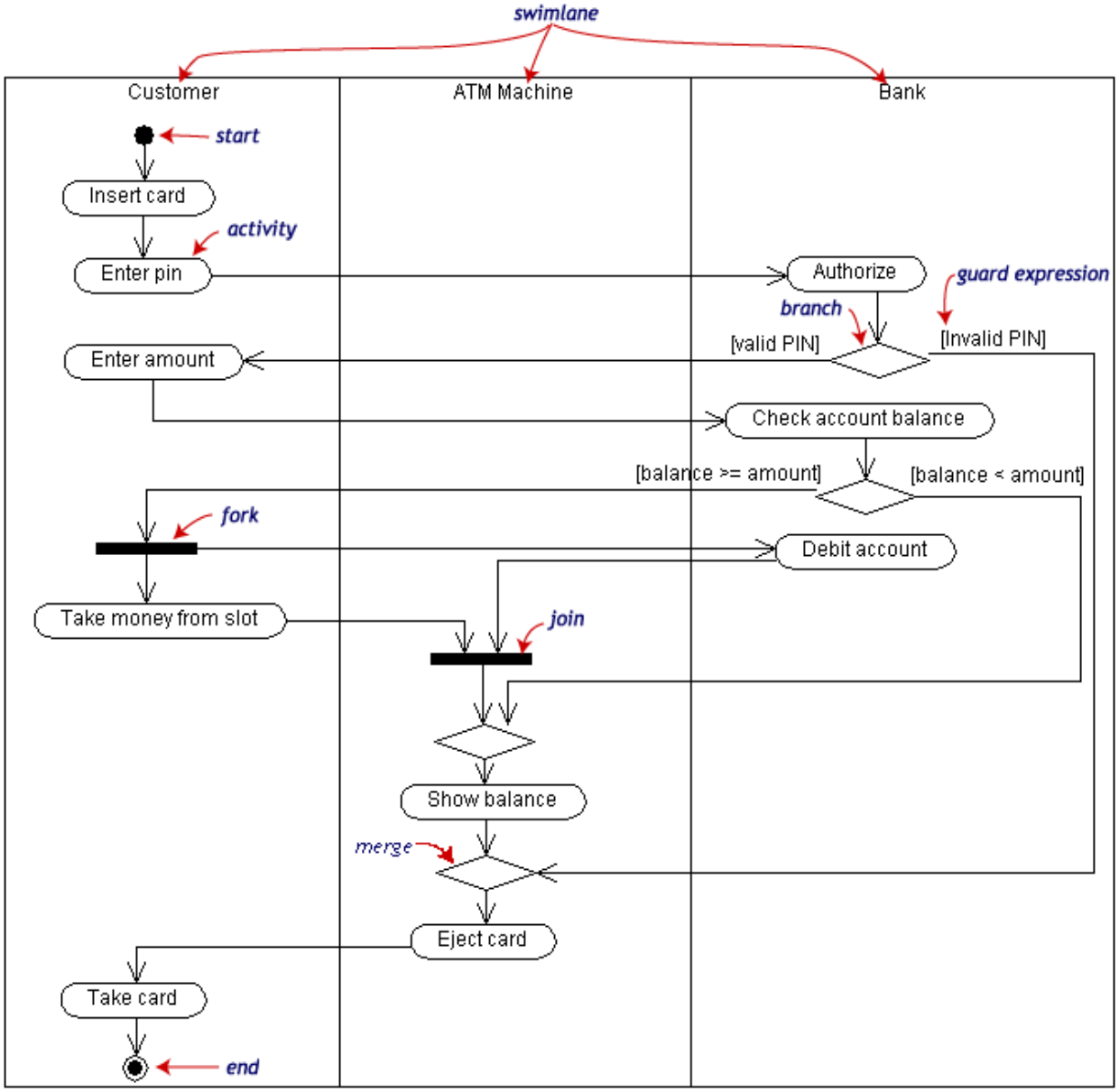
Content of Activity diagrams

- Activity diagrams commonly contain
  - Activity states and Action states
  - Transition
  - Object
- The following diagram shows the all these with well labeling:



- Diagrams are read from top to bottom and have branches and forks to describe conditions and parallel activities.
- A fork is used when multiple activities are occurring at the same time.
- The branch describes what activities will take place based on a set of conditions.
- Join is used to join multiple activities
- All branches at some point are followed by a merge to indicate the end of the conditional behavior started by that branch.

An activity diagram to Withdraw money from a bank account through an ATM.



See also UML tutorial

Statechart Diagrams

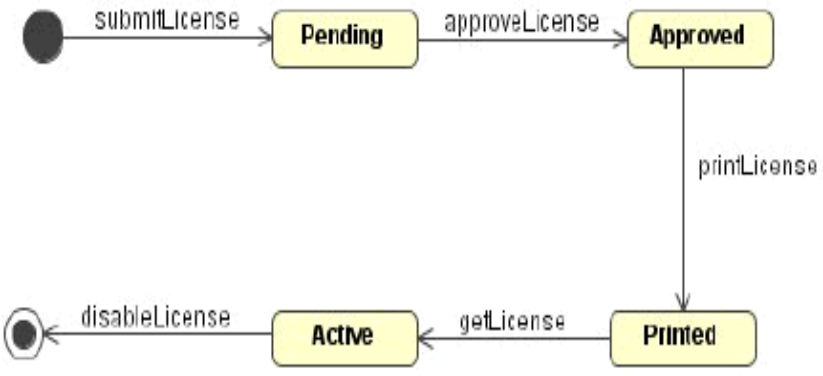
- Statechart is used for modeling the dynamic aspects of system
- A Statechart diagram shows a state machine, emphasizing the flow of control from state to state.
- A State machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- A *state* is a condition or situation in the life of an object during which it satisfies some condition, performs some activity or waits for some event.
- An ***event*** is an occurrence of a stimulus that triggers a state transition.
- The ***transition*** is a relationship between two states indicating that an object in the first state perform certain actions and enter the second state.
- An ***activity*** is ongoing nonatomic execution within a state machine.
- An ***action*** is an executable atomic computation that results in a change in state of the model or the return of a value.
- Graphically Statechart diagram is a collection of vertices and arcs

Contents and uses of Statechart diagram

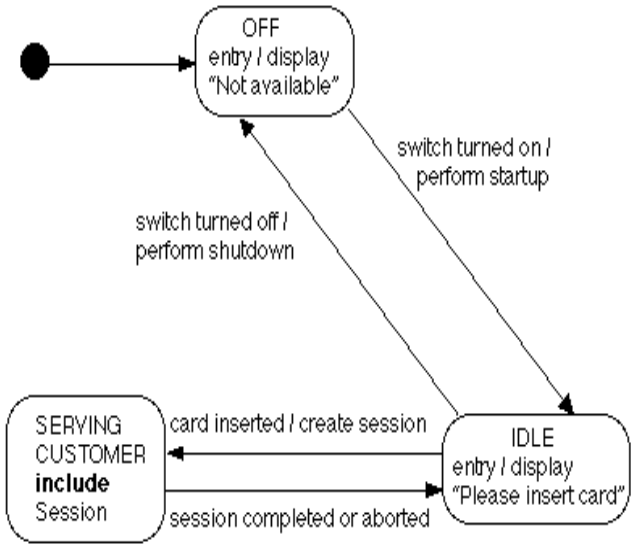
- The Statechart chart diagrams commonly contains
  - **Simple states and composite states**
  - **Transition , including events and actions** The Statechart diagram is commonly used :
    To model **Reactive Objects**

A reactive-or event-driven-object is one whose behavior is best characterized by its response to events dispatched from outside its context.

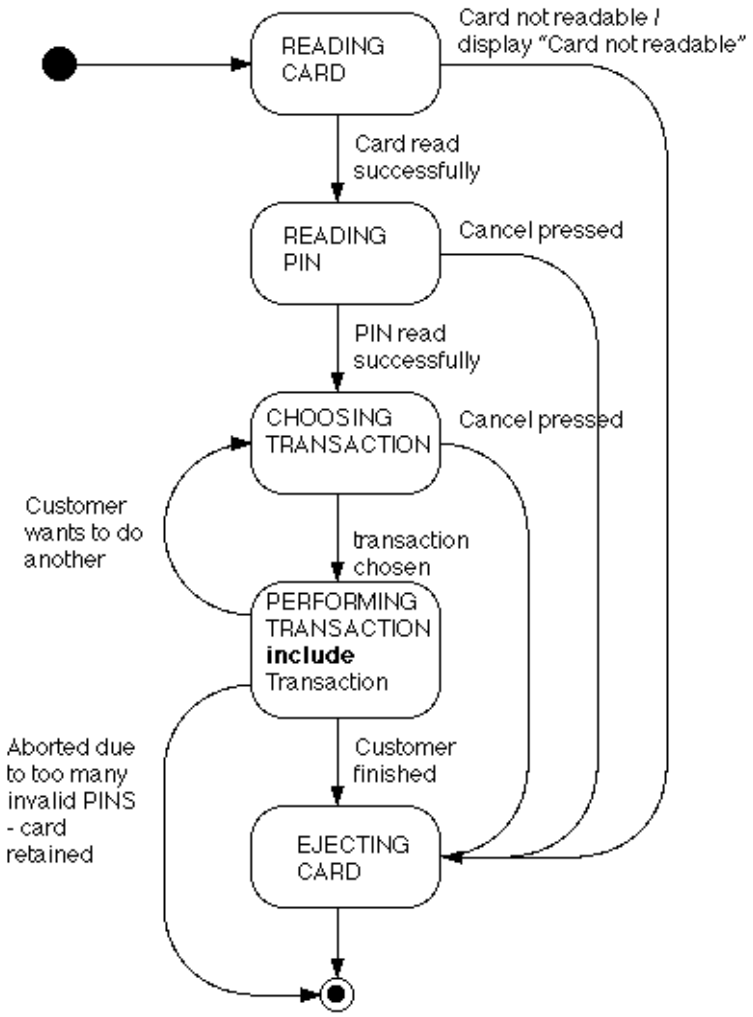
Statechart Diagram: A Simple Example

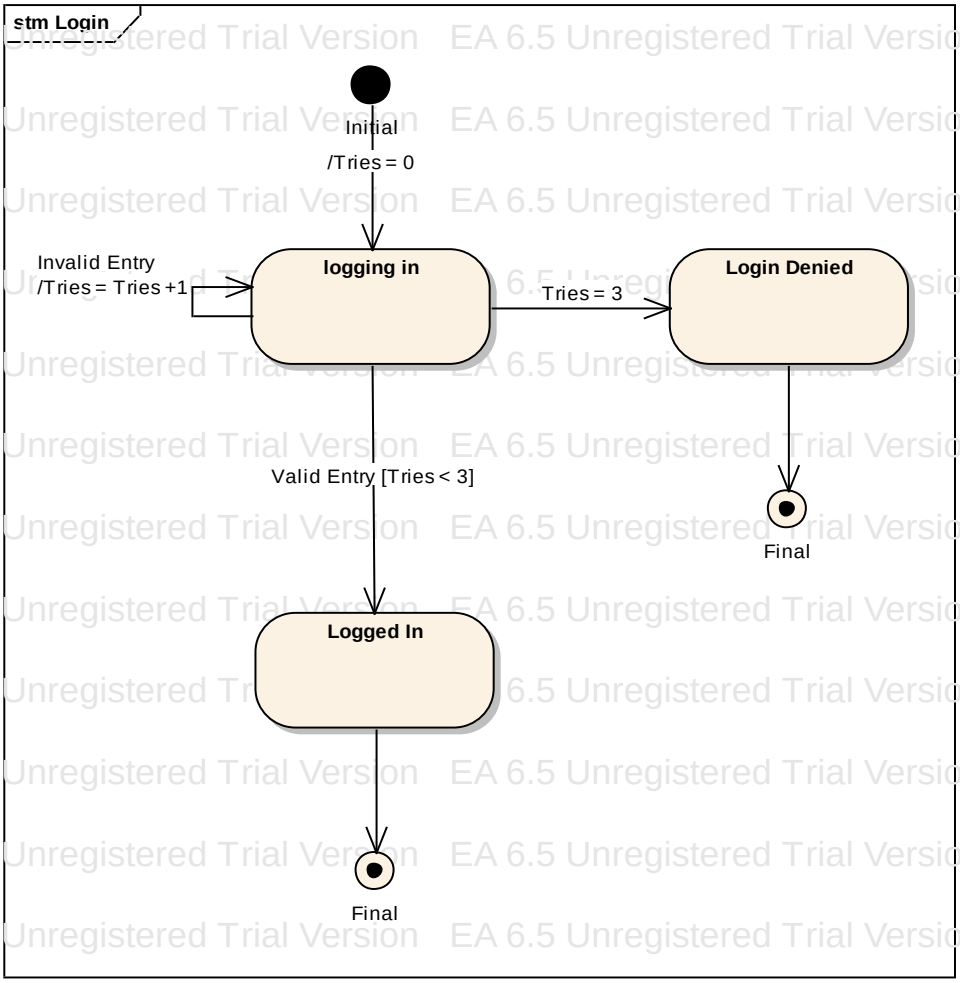


State-Chart for Overall ATM (includes System Startup and System Shutdown Use Cases)



State-Chart for One Session





Some guidelines to model a reactive object in statechart diagram are:

- Choose the context for the state machine, whether it is a class or a use case ,or the system as a whole.
- Choose the initial and final states for the object.
- Decide on the stable states of the object.
- Decide on the meaningful partial ordering of stable states over the lifetime of the object.
- Decide on the events that may trigger a transition from state to state
- Attach action on these transitions
- Check that all states are reachable under some combination of events
- Trace the state machine either manually or using tools

Note See Deployment diagram and Component diagram in UML trutorial